Logging Practices

Introduction

In the Play+ ecosystem, we believe developer tools should be invisible until needed—frictionless by default, powerful when required. This helper is based on the concept of structured, environment-aware logging.

Effective logging is crucial for observability, debugging, and understanding user behavior. The playlog helper provides a zero-setup, unified API for capturing application events. This aligns with our core design pillars by making development more Intuitive (a simple API abstracts complexity), our systems more Adaptive (clear logs help us respond to issues faster), and our codebase more Distinct (a consistent logging approach across all projects).

Package Info

The Play+ logging helper is provided via the @playplus/logging package and is included by default in the Golden Path starter kit.

Description

Package / Path	Golden Path (Recommended)
Pre-installed	/system/play.log.ts
Uplift Path	npm install @playplus/logging

Folder Reference

The logging helper and its configuration follow our standardized folder structure for core system logic.

File / Directory	Purpose & Guidelines
system/play.log.ts	The core logging service. It abstracts the underlying logging library (e.g., Pino) and handles all formatting.

File / Directory	Purpose & Guidelines
config/play.log.config.json	User-overridable configuration for log levels, PII redaction, and log transport (where logs are sent).
.env	Environment-specific variables, which can override settings in the config file.

Helper - Pillars Alignment

The playlog helper is a foundational tool that reinforces our core design pillars.

Pillar	How This Helper Aligns
Intuitive	Primary Pillar: Replaces the complexity of logging libraries with a simple, memorable API (info , warn , error).
Adaptive	Automatically adjusts its output and transport based on the environment (e.g., pretty-prints in dev, sends JSON in prod).
Distinct	Ensures a consistent structure and format for all log outputs across all Play+ applications, creating a unified signal.

Helper Overview

The playlog helper is your single, unified interface for all application logging. Its purpose is to abstract the plumbing of a production-grade logging system. Developers don't need to worry about log formatting, timestamps, redaction, or routing; they just need to call the appropriate method.

Behind the scenes, the helper automates everything:

- Structured Formatting : In production, it formats logs as JSON, ready for ingestion by services like Datadog or Sentry.
- Pretty Printing : In development, it formats logs in a human-readable, colorized way.

- Automatic Enrichment : It automatically injects context like timestamps, hostname, and application name into every log entry.
- PII Redaction : It automatically scrubs sensitive data (like passwords and tokens) based on your configuration, before the log is ever written.
- Environment-Aware Transport : It logs to the console in development but can be configured to send logs to a remote service in production.

This means a developer can simply call playlog.info() and trust that the system is doing the right thing for the current environment.

Config Options

Global configuration is managed in config/play.log.config.json and can be overridden by environment variables.

Configuration Priority

The playlog helper determines the active log level using the following priority order:

- Environment Variable : Highest priority. A variable like LOG_LEVEL=debug in your .env file or CI/CD environment will always take precedence. This is ideal for temporary debugging.
- play.log.config.json file : The project-wide default setting. This is the recommended place to set the standard for your environments.
- Built-in Default : If neither of the above is set, the helper falls back to its own internal default, which is "info" .

Configuration Details

Config Variable	Default Value	Description	Recommended Value
level	"info"	The minimum level to log. In order of verbosity: debug , info , warn , error . Can be overridden by LOG_LEVEL env var.	"debug" in dev, "info" in prod

Config Variable	Default Value	Description	Recommended Value
redact	["password", "token", "authorization"]	An array of object keys whose values will be masked (e.g., "[REDACTED]"). Uses partial matching.	Add any custom PII keys
transport.type	"console"	Where to send logs. "console" for stdout, "remote" to send to a URL.	"remote" in production
transport.apiUrl	null	If type is "remote" , this is the endpoint where logs will be sent via a POST request.	Your log ingestion URL
transport.apiKey	null	An optional API key to be sent in the Authorization header for remote transport. Use environment variables.	Your logging service API key

Helper Methods

Recommended Methods

For clarity, readability, and best practice, we recommend using these level-specific methods. They make your intent clear and allow for easy filtering in logging platforms.

Method Name	What It Does	Method Signature
debug	Logs a verbose message, useful for development and deep debugging. Hidden in production by default.	debug(message: string, context?: object): void
info	Logs an informational message, such as a user action or system event.	info(message: string, context?: object): void
warn	Logs a warning about a potential issue that doesn't prevent the application from working.	warn(message: string, context?: object): void
error	Logs an error, typically with an Error object to capture its stack trace and other details.	error(error: Error, message?: string, context?: object): void

Universal Method

The add method provides a flexible, "one-call-to-log-them-all" interface. It intelligently determines the log level based on the arguments provided. While powerful, the level-specific methods above are preferred for readability.

Method Name	What It Does	Method Signature
add	A universal logging method that handles simple messages, contextual logs, and errors intelligently.	add(messageOrError: string | Error, context?: object, message?: string): void

Usage Examples

React: Component Lifecycle and User Interaction

Angular: Service-Level Logging

Additional Info

Why We Created This Helper

Without a centralized logging helper, developers often default to using console.log , which has major drawbacks: it's unstructured, has no severity levels, cannot be configured for different environments, and reveals sensitive information in production. Setting up a proper logging library like Pino or Winston from scratch is complex and requires significant boilerplate.

The playlog helper solves this by providing a production-ready logging system out of the box. It abstracts away all the configuration and formatting, allowing developers to add meaningful, secure, and structured logs with a single line of code.

Best Practices

- Log What, Not How : Log significant events ("User created") rather than implementation details ("Function createUser was called").
- Log at the Right Level : Use error for actual errors, warn for potential issues, info for key events, and debug for temporary development tracing.
- Always Provide Context : A log message like "API failed" is useless. "API failed", { endpoint: '/users', status: 500 } is invaluable.
- Pass Full Error Objects : Always log the actual Error object in playlog.error() to automatically capture stack traces.
- Don't Log PII : Trust the redaction system, but avoid logging sensitive data in the first place where possible.

Developer Checklist

- Is my log message clear and concise?
- Have I included relevant context (like a userId or requestId)?
- Am I using the correct log level (info , warn , error)?
- For errors, am I passing the Error object to playlog.error() ?
- Have I double-checked that I'm not logging any sensitive information that isn't covered by the global redaction config?